

Existential Containers in Scala

Dimitri Racordon
dimitri.racordon@epfl.ch
EPFL, LAMP
Lausanne, Switzerland

Eugene Flesselle
eugene.flesselle@epfl.ch
EPFL, LAMP
Lausanne, Switzerland

Matthieu Bovel
matthieu.bovel@epfl.ch
EPFL, LAMP
Lausanne, Switzerland

Abstract

Type classes have been well-established as a powerful tool to write generic algorithms and data structures while escaping vexing limitations of subtyping with respect to extensibility, binary methods, and partial abstractions. Unfortunately, type classes are typically inadequate to express run-time polymorphism and dynamic dispatch, two features considered central to object-oriented systems. This paper explains how to alleviate this problem in Scala. We present existential containers, a form of existential types bounded by type classes rather than types, and explain how to implement them using Scala’s existing features.

CCS Concepts: • Software and its engineering → Polymorphism; Inheritance; Abstract data types; Object oriented languages.

Keywords: polymorphism, dynamic dispatch, type class, dependent types

ACM Reference Format:

Dimitri Racordon, Eugene Flesselle, and Matthieu Bovel. 2024. Existential Containers in Scala. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR ’24)*, September 19, 2024, Vienna, Austria. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3679007.3685056>

1 Introduction

Generic programming consists of lifting general abstractions from their concrete implementations [11]. This process requires a mechanism to write polymorphic functions and data structures, that is to write code capable of operating on different types of values. In a statically typed setting, it further requires a way of describing a common interface to these values so that a type checker may guarantee that a particular operation is sensible, regardless of the run-time type of its operands. Subtype polymorphism is one of the most

popular approaches to that goal, especially in the context of object-oriented languages.

In a system with subtyping, a type τ can be defined to be a *subtype* of type σ —typically written $\tau <: \sigma$ —so that an instance of τ may be used in all places where values of type σ are expected. The validity of the substitution hinges on the fact that τ is known to implement *at least* the interface of σ . This concept lays down solid foundations for generic programming: abstract data structures are expressed as interfaces meant to be extended by more specific definitions. For example, one may write a type `Shape` representing the concept of a polygon and have a subtype `Triangle` represent those composed of three edges. Presumably, any operation that accepts arbitrary shapes also accepts triangles.

Despite its significance for object-orientation and near ubiquitousness in contemporary programming languages, subtyping suffers from well-known limitations that hinder one’s ability to write modular yet extensible abstractions [13, 18]. Borrowing a page from functional programming, most of these shortcomings are addressed by *type classes*, such as those found in Haskell [16]. A type class describes the interface of a general concept, say a polygon, as a set of top-level functions rather than methods bound to a particular receiver. These top-level functions may be overloaded for a specific type τ , effectively specifying how τ *models* the abstract concept. Alas, while type classes elegantly fulfill the essential requirements of generic programming [15], they offer limited support for type erasure—the eliding of some type information at compile-time. Hence, it is difficult to manipulate heterogeneous collections or write procedures returning arbitrary values known to model a particular concept. Doing the same with subtyping is routine.

Existential types have been used to hide data representations since at least the 80s [7, 10] and play a pivotal role in the support of modular programs in languages like OCaml [5]. More recently, both Swift and Rust have described the signature of existential types and their implementations relying on type classes. Intuitively, given a type class instance p defining an operation $f : \sigma \rightarrow \tau$ for a value v of type σ , one can form an existential pair $\langle \sigma, \{v, p.f\} \rangle$ of type $\langle \exists X, \{X, X \rightarrow \tau\} \rangle$. This paper leverages this intuition in the context of Scala. We implement *existential containers*, a form of existentials bounded by a type class. It turns out that Scala’s type system supports all the ingredients necessary for implementing this extension efficiently. In particular, we rely on extension methods, path dependent types [14], and implicit scopes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. MPLR ’24, September 19, 2024, Vienna, Austria

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-1118-3/24/09

<https://doi.org/10.1145/3679007.3685056>

Neither type classes nor existential containers are new ideas. Our contribution is to equip Scala with an ergonomic way to use dynamic dispatch in settings relying on type classes rather than inheritance hierarchies. Section 2 motivates this goal. Section 3 introduces key features of the Scala language that are used to express type classes and can be re-used to implement existential containers, as then described in Section 4. Section 5 discusses the expressiveness of our encoding and examines its performance on different benchmarks. Section 6 positions our work in the state of the art and Section 7 concludes.

2 Dynamic Dispatch and Type Classes

Subtyping is a popular approach to support polymorphism in object-oriented programming languages. Details vary from one language to the next but in general, one defines classes to denote abstractions that are extended to model refinements, either by adding properties and operations or by specializing them. A class is *abstract* if it declares operations without an implementation and *concrete* otherwise. Abstract classes hide details irrelevant to generic algorithms.

The left-hand side of Figure 1 shows an example. Line 3 declares an abstract class `Shape` denoting arbitrary polygons with two operations. The first returns the surface area of a shape and has no implementation. Concrete types extending `Shape` must provide one. The second compares shapes and has an implementation referring to their areas. Line 9 declares a concrete subclass of `Shape` denoting squares—i.e., quadrilaterals having edges of equal lengths and equal angles—and provides an implementation for computing their area.¹ Likewise, line 14 declares a concrete subclass denoting rhombuses—i.e., quadrilaterals having edges of equal lengths with no constraint on their angles—whose implementation is omitted for conciseness. Notice that the infix operator `<` of `Shape` is a generic algorithm. It can be applied to any pair of shapes, irrespective of their representations, as demonstrated by the main function. Line 21 asserts that a square whose edges have a length of 2 is smaller than a rhombus whose diagonals have lengths of 3 and 4, respectively.

2.1 Shortcomings of Subtyping

Considering only simple examples such as the one we just discussed, we may convince ourselves that subtype polymorphism is a panacea. Abstract classes define general concepts like `Shape`, methods of such classes define generic algorithms such as `<`, and concrete subclasses model concepts such as `SquareShape`. Unfortunately, more complex situations reveal crippling limitations of this approach. While universal bounded quantification [7]—i.e., generic parameters with type bounds—solve many of them, two flaws remain.

¹We elaborate on the reason to represent the dimensions of the square as a separate object later in this section.

2.1.1 Retroactive Modeling. Looking at Figure 1, one may wonder why we defined a `SquareShape` as a class *wrapping* another data structure specifying the dimensions of a square. Why did we not define `Square` as a subclass of `Shape` directly? On closer inspection, note that we imported `Square` from a package. If we imagine that we are not the authors of that package—we may not even have access to its sources—then our choice of representation makes sense.

There is no way to *retroactively* add supertypes to a type. In other words, once a class like `Square` has been declared, its set of supertypes cannot change without also modifying the original declaration. This limitation is frustrating here because the authors of the `quadrilaterals` package have omitted the definition of an operation computing areas. While we can write it ourselves, we must use the adapter pattern to make `Square` notionally extend `Shape`.

This workaround is not ideal because it involves tedious boilerplate. Not only must we declare an additional class for each type we want to extend, we must also wrap and unwrap the contents of these classes to use methods from the original package, as line 21 demonstrates. We could improve the situation by writing a `scale` method for `RhombusShape` that forwards calls to the wrapped instance, but that would generate even more boilerplate. In a real setting, we would also have to duplicate documentation and test suites. This problem grows with the number of types we import and the number of new concepts we define. For example, we may import a `Drawable` concept from a package and a `Serializable` concept from another. Unless all shapes are drawable and serializable, we would eventually find ourselves implementing a `DrawableSerializableRhombusShape`!

2.1.2 Binary Methods. Figure 1 suggests that shapes have a total order. It may be tempting, therefore, to capture that observation in another concept that applies to shapes:

```
1 abstract class Comparable:
2   def <(s: Comparable): Boolean
```

Unfortunately, `Comparable` is a poor abstraction. To understand why, let us examine the signature of its unique operation. The operator requires that the second parameter be another instance of `Comparable`, meaning any value whose type is subtype of `Comparable` will do. However, it might not necessarily make sense to compare a shape to a character string. While there exist perfectly sensible orderings for these data types, meaning that both model a comparable concept, they do not imply the existence of a relation between a shape and a character string.

This issue is known as the *binary method problem* [6]. It states that subtyping struggles to safely express concepts having an operation of the form $\sigma \times \sigma \rightarrow \tau$. In a system where operations are written as methods of a single receiver, that means abstract concepts should avoid referring to themselves in contravariant positions. One solution is to check

```

1 import quadrilaterals.{Square, Rhombus}
2
3 abstract class Shape:
4   def area: Double
5   def <(s: Shape): Boolean =
6     area < s.area
7
8
9 final case class SquareShape(
10   dimensions: Square
11 ) extends Shape:
12   def area: Double = dimensions.side.squared
13
14 final case class RhombusShape(
15   dim: Rhombus
16 ) extends Shape: ...
17
18 @main def Main =
19   val s = SquareShape(Square(2))
20   val r = RhombusShape(Rhombus(1.5, 2))
21   assert(s < RhombusShape(r.dim.scale(2)))

```

```

1 import quadrilaterals.{Square, Rhombus}
2
3 trait Shape extends TypeClass:
4   extension (self: Self)
5     def area: Double
6     def <[S](s: S)(using S is Shape): Boolean =
7       area < s.area
8
9 given Square is Shape as SquareIsShape:
10   extension (self: Square)
11     def area: Double = self.side.squared
12
13
14 given Rhombus is Shape as RhombusIsShape: ...
15
16
17
18 @main def Main =
19   val s = Square(2)
20   val r = Rhombus(3, 4)
21   assert(s < r.scale(2))

```

Figure 1. Definitions of shapes using subtype polymorphism (left) and type classes (right) in Scala.

at run-time that the type of the method’s argument is compatible with that of its receiver. However, that comes at the expense of static type safety as some invalid applications of the operator may no longer be caught at compile-time. Another approach consists of defining an auxiliary *Comparator* class parameterized by the type of the elements:

```

1 abstract class Comparator[T]:
2   def inIncreasingOrder(l: T, r: T): Boolean

```

2.2 Type Classes

Type classes can be understood as a generalization of the trick we have just discussed. A type class describes the set of operations necessary to establish for any particular type to model a concept, thereby forming that concept’s interface. For instance, *Comparator* describes types having a strict total order. Unlike a regular abstract base type, however, a type class defines its interface using free functions rather than members of the conforming type. In our example, that means *Comparator* declares a method operating on two instances of *T* rather than assuming the receiver is one of the two operands. The receiver of that method is merely a *witness* of the conformance of *T* to the concept.

In Scala, a type class is written as a trait defining the interface of another type, similar to *Comparator*. The latter is represented by a type member—i.e., a type that depends on instances of the trait [3]—named *Self*.

```

1 trait TypeClass:
2   type Self

```

The right-hand side of figure 1 shows an example involving type classes. The trait at line 3 is a type class describing

the same concept as the abstract class on the left-hand side using method extensions. Line 9 declares an instance of the type class for *Square*. Being defined as a *given*², this instance is made implicitly available to name lookup when a method is selected on an instance of *Square*. The details of this process are discussed in the following section. For now, simply observe that the compiler considers the extensions when it resolves the entity referred to by the infix operator *<* at line 21. Line 14 similarly declares another instance for *Rhombus*.

The main function evidences the advantage of having defined the concept of shapes as a type class. We can now manipulate instances of types imported from *quadrilaterals* directly, without wrapping or unwrapping them. Further, Scala offers a pleasingly familiar programming style that supports infix application and dot notation.

2.3 Dynamic Dispatch

Our example so far has painted a rather unfavorable picture of subtype polymorphism. There is nothing, it would appear, that type classes cannot express better than inheritance. We have, however, overlooked a key feature of most object-oriented systems: *dynamic dispatch*.

Dynamic dispatch delays the selection of the definition implementing a particular generic operation until run-time, when the operation is applied. Unlike static dispatch, which makes this choice at compile-time, this technique supports type erasure and lets variables hold different types of values at run-time. For example, one can write the following using the definitions on the left-hand side of Figure 1:

² *givens* were known as *implicit*s in previous versions of Scala.

```

1 def smallest(xs: List[Shape]): Option[Shape] =
2   xs match
3     case Nil => None
4     case a :: ys =>
5       smallest(ys) match
6         case None => Some(a)
7         case Some(b) =>
8           Some(if b < a then b else a)
9 val s = smallest(List(
10   SquareShape(Square(2)),
11   RhombusShape(Rhombus(1.5, 2))
12 ))

```

This program defines an algorithm operating on a heterogeneous list of shapes, returning the smallest of its elements. Writing such a function is natural and unsurprising with subtyping. We use `Shape` to abstract over the specific types of the shapes populating the list and rely on dynamic dispatch to figure out at run-time which implementation of `area` must be called to compare instances. Writing the same function with type classes is not obvious, however, due to the lack of dynamic dispatch support. A close translation uses a generic parameter to abstract over the type of the list's contents but this approach does not accept heterogeneous lists. The fundamental issue is that we now need a witness of each element's conformance to the shape concept *and* a way to associate each element to the right witness statically. Returning an arbitrary shape is also problematic for the same reason. Not only must we return an element of the list—unless it is empty—we must also return the appropriate witness of its conformance to `Shape`. In summary, the solution conceptually looks like the following sketch:

```

1 def smallest(xs: List[(Any, AnyShapeWitness)])
2   : Option[(Any, AnyShapeWitness)] =
3   xs match
4     case Nil => None
5     case (va, wa) :: ys =>
6       smallest(ys) match
7         case None => Some((va, wa))
8         case Some((vb, wb)) =>
9           Some(
10             if wb.<(vb)(va) then (vb, wb)
11             else (va, wa)
12           )
13 val s = smallest(List(
14   (Square(2), SquareIsShape),
15   (Rhombus(1.5, 2), RhombusIsShape)
16 ))

```

The pair `(Any, AnyShapeWitness)` represents an arbitrary value along with a witness of a conformance to the shape concept. Together, these two elements let us perform dynamic dispatch, as line 10 illustrates. We call such a pair an *existential container* and spend the remainder of the paper discussing a safe and ergonomic implementation in Scala.

3 Contextual Abstractions in Scala

We ought to introduce a few features of Scala before we can delve into our implementation of existential containers, starting with context parameters. Code written with a functional flavor tends to require certain values to be passed around from one function to the next. These values represent the “context” in which a function runs, which may capture configuration settings or additional properties of other parameters. For example, consider the following:

```

1 class Tree(
2   val id: String,
3   val children: List[Tree]
4 ):
5   def depth(
6     using parent: ParentRelationships
7   ): Int =
8     parent.get(this) match
9       case None => 0
10      case Some(p) => 1 + parent.depth
11
12 given ParentRelationships =
13   computeBackwardLinks(root)
14
15 println(someChild.depth)

```

The class represents a directed tree: we can go from a node to its children but not to its parent. The method takes a context parameter denoting a table mapping each node of the tree to its innermost parent. We can use this parameter like any other but, additionally, notice it gets passed *implicitly* in recursive calls to `depth`, thus improving legibility. The value of `parent` is merely context in `depth`; it would be just a regular local binding if we had written the method with an imperative loop rather than a tail recursive call. Repeating this argument would therefore only introduce noise.

The compiler goes through a process called *implicit resolution* to identify which argument to pass to context parameters. A detailed description of this process is beyond the scope of this paper but in a nutshell, expressions live in an implicit scope populated by various constructs of the language. Those notably include context parameters and given instances as the one declared at line 12. Both may include extension methods.

Extension methods are another interesting feature, serving to retroactively “add” methods to a class without modifying its definition. They are not really members of the extended class, though. Rather, they are free functions, allowing selection with dot syntax and infix application. In the following, for instance, `encode` is added to the local scope of `Tree` so that it can be applied like any other method, as line 3 shows.

```

1 extension (t: Tree) def encode(): String =
2   val s = t.children
3     .map(_.encode()).mkString(" ")
4   s"${t.id} ${s}"

```

Moreover, extension methods can be made available through the implicit scope. By using a contextual parameter containing a member which is an extension method, the latter becomes implicitly accessible within the body of the definition. Having combined contextual abstractions and extension methods, we can now better explain the right-hand side of Figure 1, discussed in Section 2.2. Recall the definition of the type class instance for Square, repeated below.

```
1 trait Shape extends TypeClass:
2   extension (self: Self)
3     def area: Double
4     def <[S](s: S)(using S is Shape)
5       : Boolean =
6         area < s.area
7
8 given Square is Shape:
9   extension (self: Square)
10    def area: Double = self.side.squared
11
12 assert(Square(1) < Square(2))
```

In this example, the type alias “Square is Shape” expands to “Shape { **type** Self = Square }”,³ which denotes a *refinement* of the type class restricting the type of Self, the conforming type. The given declaration introduces an instance of this refinement into the implicit scope as a witness of Square’s conformance to Shape, meaning that the compiler can now use the extension methods in Shape as long as the given instance is part of the implicit scope. This witness is passed implicitly as an argument to the operator’s context parameter at line 12, allowing the compiler to resolve `s.area` at line 6.

4 Implementation

An existential container is a pair containing a value and a witness of its conformance to some type class. Expressing such a value in Scala is easy: just write “(Square(1) : Any, summon[Square is Shape] : Any)”, using `summon` to obtain a value of the given type from the implicit scope. This simple encoding, however, only gets us so far. Indeed, the erasure of the pair’s elements—notice the widening to Any—makes it impossible to select any method. Even if we could, we would still have to convince the type checker that the first element is a valid argument for these methods. One workaround could be to widen the pair’s elements to a tighter bound but this approach would rely heavily on subtyping and thus be subject to the limitations we discussed in Section 2.1. Fortunately, Scala has other tools to address these issues.

The key idea is to store the wrapped value as an instance of some *path dependent type* [14], which is already well-established as a way to implement existential types in Scala.⁴

³The `is` infix type operator is merely a standard library definition providing syntax sugaring.

⁴Note that this approach differs from Scala 2’s Existential Types, which were dropped in Scala 3[1].

```
1 /** A value together with an evidence of its
2  * type conforming to some type class. */
3 trait Container[Concept <: TypeClass]:
4   /** The type of the contained value. */
5   type Value : Concept as witness
6   /** The contained value. */
7   val value: Value
8
9 object Container:
10  /** Wraps a value of type `V` into a
11   * `Container[C]` provided a witness that
12   * `V` is `C`. */
13  def apply[C <: TypeClass](v: Any)
14    [V >: v.type](using V is C) =
15    new Container[C]:
16      type Value >: V <: V
17      val value: Value = v
```

Figure 2. Encoding of an existential container in Scala.

For example, the following trait notionally represents the type $\exists X, \{ \}$ —i.e., an existential type with no interface:

```
1 trait Existential:
2   type X
3   val value: X
4   val e: Existential = new Existential:
5     type X = Int
6     val value = 1
```

As the ascription on the declaration at line 4 suggests, `e` is an instance of `Existential`. Since `X` is a type member of that trait, we know that there exists a type `e.X` and that it is the type of `e.value`. We know nothing more. Even so, we can extract the existential value with “`val v: e.X = e.value`” and retain its type simultaneously. Crucially, we know the type of `v` *depends* on `e`, meaning that the compiler can use information stored in `e` to conclude facts about `e.X`. The last piece of the puzzle is to store a witness of `e.X`’s conformance to a type class.

4.1 Encoding

Our encoding is shown in Figure 2. The trait essentially composes two types and two values: a type `Value` with a corresponding term `value` of that type, and a type `Concept` with a witness of the conformance to that concept for the contained value. `Concept` is a simple parameter of the trait and is associated only to the type of the container being defined. `Value`, on the other end, is a type member since it *depends* on each container instance, similar to `Existential.X` in our previous example. Additionally, `Value` is context-bounded by `Concept`, meaning that instances of `Container` must provide a term of type “`Value is Concept`” named `witness`—i.e. an evidence of the conformance.

Suppose we have a value `c` of type `Container[Shape]`. We could access its parts using explicit selections and combine

the two to obtain the area of the wrapped value. Since the witness is declared as a context bound, however, it is in the implicit scope of the container's value and its selection can be elided. In effect, a selection of an extension method defined by the interface of the container—e.g., `c.value.area`—is resolved by the compiler to use the witness given within `c` implicitly. The selection of `value` can also be elided with a modest change in the compiler. Calls of the form `c.m` are transformed to `c.value.m` when `c` is a container and the selection does not type-check without being adapted. This fallback mechanism ensures that explicit selections of a container's members are still possible.

Building a container comes down to the following:

```
1 new Container[Shape]:
2   type Value = Square
3   val value = Square(2)
4   val witness = SquareIsShape
```

Here too, the witness definition can be omitted since it is declared as a context bound and `SquareIsShape` is in the context. The remaining boilerplate can be replaced by “`Container[Shape](Square(2))`” or, if `Shape` can be inferred from the expected type, “`Container(Square(2))`”. These expressions call the `apply` method of `Container`'s companion object and are equivalent to the “wrap” or “pack” primitive featured in formal definitions of existential types. This expansion is enabled by the design of the method's signature. The first type parameter list is `[C <: TypeClass]` for consistency with the trait declaration. The witness is not in the first term parameter list so that its `Self` can be constrained by the value being wrapped before looking for it in the context, thereby avoiding ambiguities. Putting everything together, we can write a well-typed version of the program we sketched at the end of Section 2:

```
1 def smallest(xs: List[Container[Shape]])
2   : Option[Container[Shape]] =
3   xs match
4     case Nil => None
5     case a :: ys =>
6       smallest(ys) match
7         case None => Some(a)
8         case Some(b) =>
9           Some(if b < a.value then b else a)
10  val s = smallest(List(
11    Container(Square(2)),
12    Container(Rhombus(1.5, 2))
13  ))
```

The method accepts a heterogeneous list of shapes represented and returns the smallest unless the list is empty. It is called at line 10, where its argument is constructed by wrapping different shapes along with their respective conformances to `Shape`. The body of the method reads almost the same as if we had used subtyping with one difference. We must write `a.value` rather than just `a` for the argument of the operator `<` at line 9. The next section explains why.

4.2 Opening Containers

Perhaps unintuitively, `Container[C]` is not itself a type conforming to `C` since an existential container does not conform to its own bound. Recall the binary method problem to understand why. Imagine we define a type class `C` modeling comparable values and consider two values of types `Int` and `String`, respectively, each conforming to `C`. We could then create two different existential containers wrapping 42 and “forty-two”, respectively. Although both instances would have type `Container[C]`, they should not be comparable, and therefore `Container[C]` cannot conform to `C`.

Coming back to the above example, we can now observe that a call of the form `b < a` would not type check. Indeed, the operator requires the witness of a conformance to `Shape` for its right operand (see Figure 1) but such a witness does not exist for `Container[C]`. By passing `a.value` however, we pass an argument of type `a.Value`, of which the implicit scope includes the witness given in `a`. More formally, accessing `a.value` opens the existential while retaining the type `a.Value`, effectively keeping an anchor—the path to the scope of the witness—to the interface of the type class. This example showcases the power of path dependent types. The ability to use an opened existential without escaping static type safety—i.e., without unchecked type coercions—does not require any amendment to the type system.

4.3 Using Containers over Generic Parameters

While the ability to open an existential grants us enough expressiveness to call a generic function with a context bound, we could instead define a non-parametric version of our operator, using existential containers, to apply `b < a` without any explicit opening. Taking another look at the signature, we notice that we actually care neither about which specific type of shape we receive nor about the accompanying type class instance, as long as get to call `area` on an object representing a shape. An existential container gives the same guarantees with a shorter spelling:

```
1 // Existential version
2 def <(s: Container[Shape]): Boolean =
3   area < s.area
4 // Parametric version
5 def <[S](s: S)(using S is Shape): Boolean =
6   area < s.area
```

This existential definition is arguably simpler than its parametric counterpart. The ascription on `s` in the former completely defines the type of the arguments that the method expects. In contrast, the type of `s` in the latter is the composition of three elements: the generic parameter declaration, the context parameter, and the ascription itself. While a generic parameter may be sometimes required, e.g., to address binary methods, an existential container is more appropriate when a method simply expects a value of *some* type with a compatible type class instance.

5 Evaluation

This section discusses the applicability of our implementation. We start by studying the expressiveness of existential containers in a use case more sophisticated than the examples we have seen so far. We then measure the performance of our implementation in a realistic setup and compare it with subtyping before discussing some limitations.

5.1 Expressiveness

So far we have only examined type classes with a single parameter: the conforming type. It is likely, however, to encounter type classes having more than one parameter. We call those *associated types*, following the steps of Järvi et al on generic programming [9]. To illustrate, consider the following example.

```
1 trait Archivable extends TypeClass:
2   type Encoder
3   type Decoder
4   extension (self: Self)
5     def encode()(using encoder: Encoder): Unit
6     def decode()(using decoder: Decoder): Self
```

Archivable defines an interface for serializable objects by means of two methods; one for serialization and the other for deserialization. The signature of these methods refers to Encoder and Decoder, which are associated types declared as path dependent types.

We can certainly create a heterogeneous list of archivable objects, as we have already demonstrated. To write a function serializing the contents of this list, however, we need a more precise type since each element could conform to Archivable for a different type of encoder. In other words, we must constrain each element's witness to have the same associated type without constraining the type of their value. Fortunately, a simple refinement suffices:

```
1 def encodeAll[E](
2   xs: List[
3     Container[Archivable { type Encoder = E }]
4   ]
5 )(using E): Unit =
6   xs.forEach(x => x.encode())
```

Two important observations can be made on this example. First, it shows that our encoding of implementation containers fits perfectly into Scala's existing support for type refinements, letting us attach equality and/or conformance constraints to associated types. That is an essential requirement for generic programming [15]. Second, notice that the signature of encodeAll does not have to mention anything about the *decoder* of its arguments. Only the relevant parameters of the interface need to be specified; the others can remain abstract. That is in contrast with approaches based on universal bounded quantification which typically require all parameters to appear in type constructors, along with a sensible bound. In some instances, finding such a bound is,

in fact, impossible and one must resort to Java-like wildcards, which come with well-known soundness issues [4].

5.2 Performance

The main question concerning the performance of our implementation is whether calling methods on existential containers incurs significant overhead compared to traditional dynamic dispatch. We ran two benchmarks to test this overhead. The first is a micro benchmark focusing only on the dispatch time of an operation. The second is a macro benchmark aiming at measuring the overhead in a realistic scenario. We implemented two versions of each benchmark: one uses traditional subtyping and the other uses existential containers. We also implemented similar benchmarks in Swift to provide an inter-language perspective. The source code for these programs is available as a companion artifact. Hereafter, we present and discuss measures from 10,000 iterations across 4 runs for each version⁵.

5.2.1 Micro Benchmark. The results of the micro benchmark are presented in Figure 3. We observe that dispatching an operation through existential containers in Scala is about twice as slow as traditional virtual method dispatch. This can be explained by the extra pointer indirection. On the JVM, the runtime increases with the number of classes. For the Existential version, it ranges from approximately 5 ms for 2 classes to around 8 ms for 10 classes, and stays stable for a higher number of classes. For the Inheritance version, we observe similar curves, ranging from about 2.5 ms to 5 ms. The Swift measures are stable across the number of distinct classes, which is expected due to the lack of just-in-time compilation. Using inheritance in Swift is especially slow due to the reference counting overhead for classes.

5.2.2 Macro Benchmark. To test the performance of existential containers in a real-world scenario, we have implemented a small (≈ 1500 lines of code) 3D collision engine, which we use to detect object occlusion from a particular point of view. The shapes of different 3D objects are approximated by idealized mathematical volumes, such as spheres and cuboids, positioned randomly within an imaginary bounding box. Occlusion is then detected by shooting a ray in the direction of each object and checking whether another object lies in the way. Each collision shape has its own data representation—e.g., a sphere is merely a radius—and implements its own collision detection algorithm. To select

⁵We ran both the *micro benchmark* and the *macro benchmark* 4 times, alternating versions. Each run comprised 2,500 warm-up iterations and 2,500 measurement iterations, totaling 10,000 measurements. The benchmarks were executed on a MacBook Pro with a 2.6 GHz 6-Core Intel Core i7 processor, using Swift 5.10.0.13, Swift Benchmark 1.23.1, a nightly version of Scala 3.5.0 and JMH 1.37 through sbt-jmh 0.4. The "openjdk" JVM is Adoptium 1.21.0.3, and "graal" is GraalVM 21.0.2. The benchmark results were processed and visualized using the Seaborn, Matplotlib, and Pandas Python libraries.

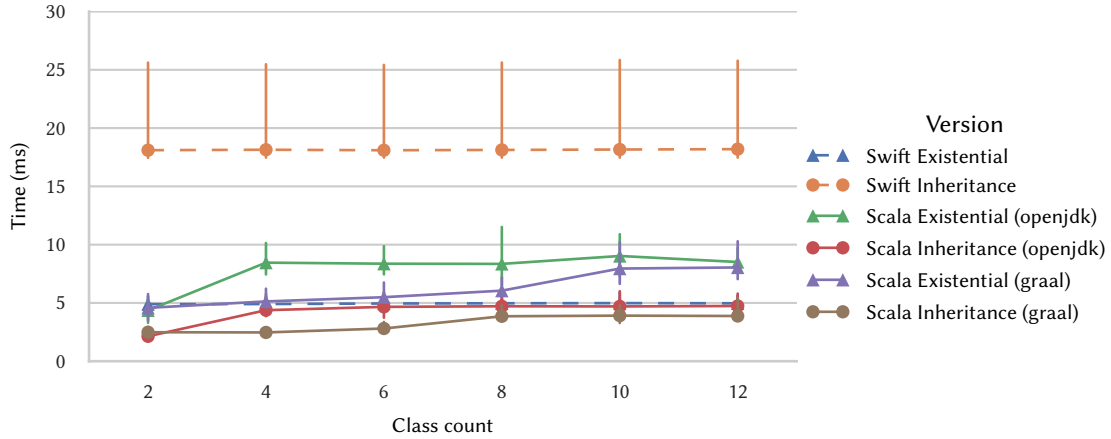


Figure 3. Micro benchmark results. This benchmark measures the time to apply a simple operation (adding the value of a class field to a constant) to 10 million objects in a contiguous array. The operation is modeled using a type class and existential containers in the Existential versions, while it is modeled using a virtual method in the Inheritance version. The graph shows the runtime in milliseconds for the 10 million operations, as a function of the number of different object types present in the array. The error bars show the 99th percentile, while the markers represent the measurement means. Lower is better.

the right implementation at run-time for each element of the heterogeneous collection of shapes, we use existential containers in one version and classes with inheritance in the other. The program terminates after a ray has been shot at every object. Looking at the results, we observe that the overhead of the existential version in Scala is 6% for OpenJDK and 2% for GraalVM.

5.3 Limitations

An important limitation of our encoding is that an existential container can only have exactly one bound; that is, it can only wrap a single evidence. As a consequence, one cannot easily express a type representing, for instance, any shape that is also archivable. One workaround is to define a new type class that extends both concepts, which in Scala can be done with an intersection type. This approach has one problem, however. It induces significant boilerplate to provide and manipulate witnesses, as it is not obvious that a type conforming to both T and U automatically conforms to a type class $T \& U$ without manually aggregating witnesses. Further, that has to be done for all different pairs of type classes for which an aggregate is needed.

6 Related Work

JavaGI [17] is an extension of Java that supports *interface types*, a generalization of Java’s interface supporting retroactive modeling, type-safe binary methods, implementation inheritance, and dynamic dispatch. These features closely

match the use cases of type classes in Scala.⁶ A significant difference is that, unlike a type class, an interface type defines its operations in terms of a receiver—i.e., the “this” parameter in Java-family languages—rather than methods of an ad-hoc object. This approach is more in line with classical interfaces: whereas a type class instance *witnesses* the conformance of a type to some concept, an interface type instance *is* a model of some concept. Hence, interface type can be used naturally to express polymorphic functions and data structures, whereas one must use existential containers to achieve the same with type classes. In contrast to existential containers, however, JavaGI’s interfaces cannot be used as types if they define associated types.

Similar efforts include Genus [19], another extension of Java that borrows from early designs of C++ concepts [8, 15] to express type classes as sets of type constraints. Unlike JavaGI, Genus supports non-uniquely witnessed conformances along with language mechanisms to resolve ambiguities. Similar expressiveness can be achieved naturally with our encoding in Scala by passing arguments to context parameters explicitly.

As we have already mentioned, Swift advocates for a type class oriented programming style and uses existential containers for dynamic polymorphism. While our approach requires minimal change to the Scala language and its compiler, Swift’s built-in support for existential types offers advantages with respect to ergonomics. Specifically, the subtyping

⁶The intersection between Scala’s type classes and JavaGI’s interfaces is even larger and covers features we have not discussed in this paper, notably including static methods and conditional conformance.

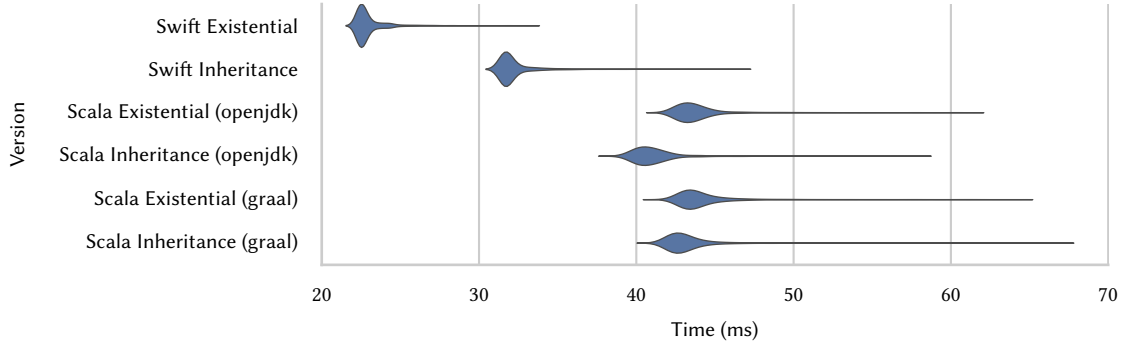


Figure 4. Graph of the *macro benchmark* results. This benchmark measures the time to shoot a ray at every shape in a contiguous array of 1000 shapes. As in the micro benchmark, operations are modeled using a type class and existential containers in the Existential versions, while they are modeled using a virtual method in the Inheritance version. The graph uses a violin plot to compare the distributions of the runtimes (in milliseconds) for each version. Closer to the left is better.

relationship can be extended so that a type T can be considered subtype of any P , where P is a type class to which T conforms. As a result, the wrapping/unwrapping of values into/from existential containers interacts more organically with the language’s syntax. Further, Swift can express compositions of type classes, therefore providing built-in support for existential containers bound by more than one type class. This system is formally described by Racordon and Buchs [12], though their work does not consider constraints on associated types.

Rust also supports existential containers as a built-in feature, where `dyn P` is similar to Swift’s `any P`. However, it imposes heavy restrictions on the definition of the traits that can be used with `dyn`. In particular, these cannot contain static or generic methods [2]. Our approach does not suffer from any of these limitations.

7 Conclusion

We have presented an extension of the Scala programming language to support *existential containers*, a form of existential types bounded by type classes rather than types. These containers are represented as a dependent pair $\langle v, w \rangle$ where w is an instance *witnessing* the conformance of v ’s type to some type class. Our approach exploits Scala’s existing features to not only express such a dependent pair but also use the operations defined by the type class in a natural way, which has two advantages. First, the soundness of the theoretical foundations upon which these features are built [4] extends to our work, which requires no change in the metatheory of the language. Second, the fact that our encoding only involves a modest desugaring transformation in the compiler gives us confidence that extending Scala with existential containers does not introduce bugs due to unforeseen interactions with other features.

The two essential features we have used, namely path dependent types and context parameters, are not exclusive to Scala. Hence, our approach could apply to a number of other programming languages.

We have motivated the need for existential containers, discussed their practicality, and studied their efficiency. While our examples demonstrate the gain in expressiveness, performance results reveal that dynamic dispatch through existential containers incurs noticeable overhead over traditional virtual dispatch. Our experiments show that such an overhead might be affordable in applications not bottlenecked by method dispatch. Nonetheless, these results suggest that more engineering effort should be spent on optimizing calls to extension methods. Other future works include an extension of our encoding to support existential containers bound by more than a single type class. We believe that existential containers open the door to alternative ways to design libraries that may suffer less from the extensibility limitations of subtyping and plan on proposing our encoding and compiler changes to be included in future versions of Scala.

Acknowledgments

This project has been funded by the Swiss National Science Foundation project entitled “Capabilities for Typing Resources and Effect” (TMAG-2_209506/1).

References

- [1] [n. d.]. Dropped Scala 2 Existential Types. <https://docs.scala-lang.org/scala3/reference/dropped-features/existential-types.html> Accessed 16 July 2024.
- [2] [n. d.]. Object Safety—The Rust Reference. <https://doc.rust-lang.org/reference/items/traits.html#object-safety> Accessed 16 July 2024.
- [3] Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of path-dependent types. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 233–249. <https://doi.org/10.1145/2660193.2660216>

- [4] Nada Amin and Ross Tate. 2016. Java and scala's type systems are unsound: the existential crisis of null pointers. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 838–848. <https://doi.org/10.1145/2983990.2984004>
- [5] Clément Blaudeau, Didier Rémy, and Gabriel Radanne. 2023. Retrofitting OCaml modules. In *Journées Francophones des Langages Applicatifs (JFLA)*. 59–100.
- [6] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. 1995. On Binary Methods. *Theory and Practice of Object Systems (TAPOS)* 1, 3 (1995), 221–242.
- [7] Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computer Surveys* 17, 4 (1985), 471–522. <https://doi.org/10.1145/6041.6042>
- [8] Douglas P. Gregor, Jaakko Järvi, Jeremy G. Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. 2006. Concepts: linguistic support for generic programming in C++. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Peri L. Tarr and William R. Cook (Eds.). ACM, 291–310. <https://doi.org/10.1145/1167473.1167499>
- [9] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. 2005. Associated types and constraint propagation for mainstream object-oriented generics. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 1–19. <https://doi.org/10.1145/1094811.1094813>
- [10] John C. Mitchell and Gordon D. Plotkin. 1988. Abstract Types Have Existential Type. *ACM Trans. Program. Lang. Syst.* 10, 3 (1988), 470–502. <https://doi.org/10.1145/44501.45065>
- [11] David R. Musser and Alexander A. Stepanov. 1988. Generic Programming. In *Symbolic and Algebraic Computation, International Symposium (ISSAC) (Lecture Notes in Computer Science, Vol. 358)*, Patrizia M. Gianni (Ed.). Springer, 13–25. https://doi.org/10.1007/3-540-51084-2_2
- [12] Dimi Racordon and Didier Buchs. 2020. Featherweight Swift: a Core calculus for Swift's type system. In *ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, Ralf Lämmel, Laurence Tratt, and Juan de Lara (Eds.). ACM, 140–154. <https://doi.org/10.1145/3426425.3426939>
- [13] John C. Reynolds. 1978. *User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction*. Springer New York, New York, NY, 309–317. https://doi.org/10.1007/978-1-4612-6315-9_22
- [14] Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 624–641. <https://doi.org/10.1145/2983990.2984008>
- [15] Jeremy G. Siek and Andrew Lumsdaine. 2005. Essential language support for generic programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 73–84. <https://doi.org/10.1145/1065010.1065021>
- [16] Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 60–76. <https://doi.org/10.1145/75277.75283>
- [17] Stefan Wehr. 2010. *JavaGI: a language with generalized interfaces*. Ph.D. Dissertation. University of Freiburg. <http://www.freidok.uni-freiburg.de/volltexte/7678/>
- [18] Stefan Wehr and Peter Thiemann. 2011. JavaGI: The Interaction of Type Classes with Interfaces and Inheritance. *ACM Transactions on Programming Languages and Systems* 33, 4 (2011), 12:1–12:83. <https://doi.org/10.1145/1985342.1985343>
- [19] Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. 2015. Lightweight, flexible object-oriented generics. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, David Grove and Stephen M. Blackburn (Eds.). ACM, 436–445. <https://doi.org/10.1145/2737924.2738008>

Received 2024-05-25; accepted 2024-06-24